Answering XML Queries Using Indexes

Kam-Fai Wong Jeffrey Xu Yu Nan Tang

Department of Systems Engineering and Engineering Management
The Chinese University of Hong Kong, Hong Kong, China
{kfwong, yu, ntang}@se.cuhk.edu.hk

Abstract

The problem of answering XML queries using indexes is to find efficient methods for accelerating the XML query with pre-designed index structures over the XML database. This problem received increasing interests and have been lucubrated in recent years. Regular path expression is the core of the XML query languages e.g., XPath and XQuery. Most of the state-of-the-art XML indexes, therefore, hammer at how to efficiently answer the path-based XML queries. This paper surveys various approaches to indexing XML data proposed in the literature. We give a step by step analysis to show the evolution of index structures for XML path information, based on tree structures or more commonly, directed labeled graphs. For each approach, we first present the specific issue it aims to tackle, and then the proposed solution presented. Furthermore, construction, physical data storage and maintenance costs, are analyzed.

1 Introduction

Recent interests and development in Extensible Markup Language (XML) has made it the *de facto* standard for data exchange among different applications over the Internet. For instance, in electronic data interchange, e-commerce, electronic publishing, and database applications, etc. XML is a form of Semistructured Data [**?**]. Its powerful and flexible data exchange model, based on the object exchange model (OEM) [**?**], has brought many interesting challenges to the database community, of effective management and query of XML data.

The explicitly specified schema constraints in both traditional relational and object-oriented database systems are inappropriate for the irregular nature of semistructured XML data. This issue leads to the proposal of a large list of XML storage models. SOTRED [**?**], Edge [**?**], MOA [**?**], Monet [**?**], XRel [**?**], VXMLR [**?**], XParent [**?**] and XISS/R [**?**] are proposed to take advantage of the mature relational technology. An alternative approach is to build a specialized data manager that contains a semistructured data repository at its core. This is referred to as native XML database. Lore [**?**] (for *Lightweight Object Repository*), Timber [**?**], XBase [**?**] and some industrial products such as Tamino [**?**] and XYZFind [**?**] adopt this approach. Compared with relational systems, native XML databases give us more opportunities to optimize the data storage, access methods and query processing to achieve better system performance.

A novel data model always requires a query languages which takes advantage of the storage features of that model. The XML data model is of no exception. A large number of query languages have been developed such as UnQL [**?**], Lorel [**?**], XML-QL [**?**], Strudel [**?**], Quilt [**?**], XPath [**?**] and XQuery [**?**]. Regular path expressions [?, ?, **?**, **?**] are the bases of these languages.

Database system indexes play an important role in accelerating user queries. The performance benefit of an index is maximized when the query matches the index definition. And the target of an index design is to make it

fit for the queries rather than let queries conform to the structure. XML path indexes for regular path expressions could greatly improve the performance of XML databases. This paper surveys approaches to indexing XML data proposed in the literature. For each approach, we present the specific issue it aims to tackle. Moreover, we picture its structure and show how to navigate XML queries. There are many criteria to compare the performance of indexes. In this paper, we choose three pivotal criteria for uniform comparison. The first criteria is construction cost. It is to show the time complexity of indexes' construction. If an index structure could efficiently perform query operation, but the construction cost is high, the index structure itself must be re-designed or an effective construction algorithm must be found. Navigational index structures are often too complex or large to be held in main memory. Therefore, all index structures must be designed to trade off between storage space and search performance. The different index structures vary widely in their storage consumption. If the index structure is larger than original data, query over it may even be slower than directly on the data graph. In this case, the index structure is meaningless. So the second criteria adopted is storage space, which is for the index structures' space complexity. Update cost is another important criteria. It is essential that an index structure can be modified to reflect the changes made to its corresponding data graph. Some index structures support incremental update, and reflect changes only to the related portions of the database; and others make the system to rebuild the index completely, e.g. when a new node is inserted at a specific position. As a consequence, the latter do not support dynamic databases. Furthermore, we will analyze how different approaches support various queries.

The rest of this paper is organized as follows. In Section 2 we use actual examples to analyze the motivation for XML path indexes. Next, in Section 3 we give the terminologies used for different approaches throughout this paper. Section 4 surveys some classical approaches and Section 5 compares and analyzes these approaches. Section **??** introduces some related works. Finally Section 6 concludes this paper.

2 Illustrative Examples

We first outline the problem of answering XML queries using indexes.

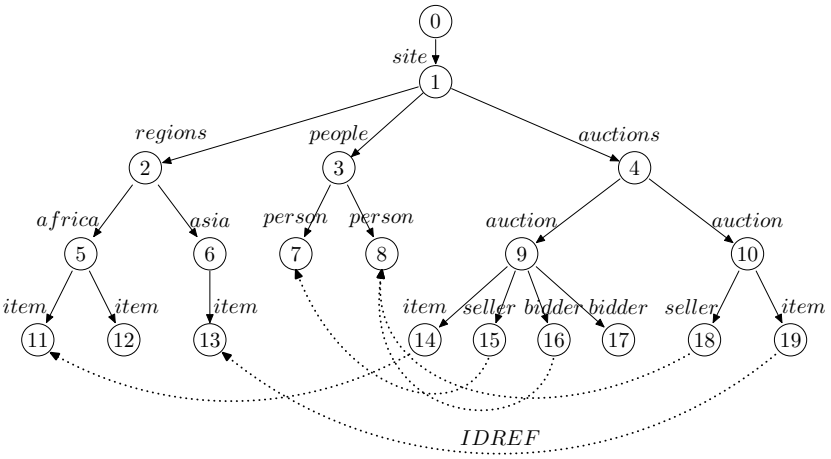


Figure 1. A sample native XML database

Figure 1 presents a small XML database, representing an imaginary auction site. The DTD adopted is similar to that in [**?**]. XML database could be looked as a graph, or more concise, a data tree without considering *id/idref* attributes. Table **??** presents the same XML database represented in a relational table. The attribute "Label" is the tag for edge, the attribute "Ordinal" is the order upon the same tagged edge, the attribute "Source" is the node ID of edge's start node and "Target" is the node ID of edge's terminative node.

Considering the following query which retrieves all sellers appearing in the auction: */site/auctions/auction/seller*.

Label	Ordinal	Source	Target
"site"	1	0	1
"auctions"	1	1	4
"auction"	1	4	9
"auction"	2	4	10
"seller"	1	9	15
"seller"	2	10	18
...	...	...	...

Table 1. A relational XML database

Here, the slash "/" represents the parent-child relationship. To evaluate this query without any path indexes in a native XML database, we need to traverse the whole tree structure. For instance, starting from the edge $site$, we search its children edge $auction$, until we get all the element via label $seller$. In relational database, one approach is to join the tuples T satisfying $T_{Label="site"}$ with tuples Q satisfying $Q_{Label="auctions"}$ upon the condition $T_{\cdot Target} = Q_{\cdot Source}$. The intermediate results are then joined with tuples S satisfying $S_{Lable="auction"}$ and so on. We get the final results until all the partial joins are performed. Both approaches are expensive. If we have path indexes for the path */site/auctions/auction/seller*, this query will be greatly expedited.

The above case is a path query starting from the root, and only follows the child axis "/" [?]. This query requires that users know the exact structure of the tree. However, this is impractical. Consider another query */site//seller* which also retrieves all the sellers in the auctions. Here, the double slash "//" represents the ancestor-descendant relationship. This query is more complex because in native XML database, we need to traverse all the descendants of nodes via labels $regions$ and $people$, and in relational database implementation, we must join even many more insignificant relations.

Moreover, you cannot expect that all queries starting from the root. Consider the case *//auctions/auction/seller* or *//auction//seller*. We may first scan the elements named $auctions$ and then traverse their subtrees. Note that some elements may be nested themselves, which will then increase the execution complexity. What we have shown are just the simplest path cases containing only single path in the queries. If the queries contain one branch like *//auction[//seller]//bidder*, this kind of queries will require more time with the absence of indexes. Furthermore, one query could include multiple branches, predicates or even contain wildcards like $*$, $?$, $+$. Each *per se* is a tough task. Efficient execution of these examples requires path indexes.

3 Preliminaries

3.1 Data Model

XML is modeled as a labeled directed graph, which is similar to the OEM model [?]. Nodes correspond to objects in the database and edges to attributes. In particular, two attributes ID and IDREF, provide node references and using them, XML data can be represented as a graph structure. Assume an infinite set $\mathcal{D}$ of data values and an infinite set $\mathcal{N}$ of nodes.

DEFINITION 1. *(Data Graph)* The structure of XML data is represented by the directed labeled edge graph G_x. $G_x = (V, E, root)$, where $V \in \mathcal{N}$ is a finite set of nodes; $E \subseteq V \times \mathcal{D} \times V$ is a set of labeled edges, $root \in V$ is the root of G_x. Each node in G_x has a unique node identifier (nid). We will often refer to such a data graph as a XML database. ■

As shown in Figure 1, the reference relationship *IDREF* is represented using a dashed edge. The numbers in-

side each circle represent node ID, ie. nid. The string near the edge is the label.

DEFINITION 2. *(Extent)* In G_x, an $extent$ is a set of t of nids such that there exists some label path l of s where $t = \{o | l_1.o_1.l_2.o_2 \dots l_n.o$ is a data path instance of $l\}$. That is, the extent set t is the set of all nodes that can be reached by traversing a given label path l of s. We also say that t is "the extent set of l in s," and we write $t = T_s(l)$. We say that l reaches any element of t, and likewise each element of t is reachable via l. ■

For instance, the $extent$ set of *site.auctions.auction.bidder* in Figure 1 is $\{16, 17\}$. Note that two different label paths may share the same target set. For example, the set $\{7, 8\}$, is the $extent$ set of both *site.people.person* and *site.auctions.auction.IDREF*.

3.2 Path Model and Query Model

DEFINITION 3. *(Label Path)* A *label path* of a node o in G_x is a sequence of one or more dot-separated labels, $l_1.l_2 \dots l_n$, such that we can traverse a path of n edges $(e_1 \dots e_n)$ from o where edge e_i has a label l_i. ■

In Figure 1, $africa.item$ and $asia$ are both valid label paths of node 2. In XML data, queries are based on label path such as $//auction/seller$.

DEFINITION 4. *(Data Path)* A *data path* of a node o in G_x is a dot-separated alternating sequence of labels and nids of the form $l_1.o_1.l_2.o_2 \dots l_n.o_n$, such that we can traverse from o a path of n edges $(e_1 \dots e_n)$ through n nodes $(x_1 \dots x_n)$, where the edge e_i has the label l_i and the node x_i has the nid o_i. ■

In Figure 1, $africa.5.item.11$ and $asia.6$ are data paths from node 2.

DEFINITION 5. *(Instance)* A data path d is an $instance$ of a label path l if the sequence labels made from d by eliminating nids is equal to l. ■

Again in Figure 1, $africa.5.item.11$ is an instance of $africa.item$ and $asia.6$ is an instance of $asia$.

DEFINITION 6. *(Label Path Containment)* A label path $A = a_1.a_2 \dots a_n$ is contained in another label path $B = b_1.b_2 \dots b_m$ if we have $a_1 = b_i$, $a_2 = b_{i+1}$,$ldots$, $a_n = b_{i+n-1}$ where $1 \leq i$ and $i + n - 1 \leq m$. When A is contained in B, we could also call that B contains A or A is a $subpath$ of B. Furthermore, when A is a subpath of B and $m = i + n - 1$, we call that A is a suffix of B. ■

For example, the label path $item$ is a subpath of $africa.item$. And, the label path $item$ is a suffix of $africa.item$.

We assume a set of base predicate $p_1, p_2 \dots$ over the domain of values $\mathcal{D}$, and denote with $\mathcal{F}$ the set of boolean combinations of such predicates. We assume that we have effective procedures for evaluating the truth values of sentences $f(d)$ and $\exists x.f(x)$, for $f \in \mathcal{F}$ and $d \in \mathcal{D}$.

DEFINITION 7. *(Path expressions)* Path expression P, over formulae in $\mathcal{F}$:$P ::= \emptyset \mid \epsilon \mid f \mid (P|P) \mid P.P \mid P.*$. We denote with $L(P)$ the regular language defined by P, and with $W(P)$ the set of all words $w = a_1 \dots a_n$ in D^*, such that there exists a word $w' = f_1 \dots f_n \in L(P)$ and $f_i(a_i)$ holds for all $i = 1 \dots n$. (i.e. the set of words obtained by replacing each formula by some value that satisfies it). It is easy to see that the languages defined by path expressions are closed under intersection and that the emptiness problem for $W(P)$ is decidable. ■

Given a data graph G_x and a path $p = v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \cdots v_{n-1} \xrightarrow{a_n} v_n$ in G_x. We say that p matches the

path expression P iff the word $a_1 \ldots a_n$ is in $W(P)$.

3.3 Bisimulation

Many existing index structures for XML are based on the notion of bisimulation. In this section we introduce some definition about it.

DEFINITION 8. *(Bisimulation)* $\approx$, the bisimulation on G_x, is a symmetric binary relation of two nodes. We say that two data nodes u and v are bisimilar ($u \approx v$), if (i) u and v have the same label; (ii) if u' is a parent of u, then there is a parent v' of v such that $u' \approx v'$, and vice versa. ■

Two nodes u and v in the data graph G_x are bisimilar, denoted as $u \approx_b v$, if there is some bisimulation such that $u \approx v$. For example, in Figure 1, nodes 15 and 18 ($seller$) are bisimilar, while nodes 11 and 12 are not bisimilar, because node 11 has an incoming reference edge but node 12 doesn't. We can easily come to the conclusion by induction that if two nodes are bisimilar, the sets of paths coming into them are the same.

DEFINITION 9. (k-*bisimilarity* $\approx^k$) k-*bisimilarity* is defined inductively:

1. For any two nodes, u and v, $u \approx^0 v$ if and only if u and v have the same label.
2. $u \approx^k v$ iff $u \approx^{k-1} v$ and for every parent u' of u, there is a parent v' of v such that $u' \approx^{k-1} v'$, and vice versa. ■

If nodes u and v are *k-bisimilar*, then the set of label-paths of $length \leqslant k$ into them is the same. *k-bisimilar* is transitive. Consider three nodes *x, y, z*. If $x \approx^k y$ and $y \approx^k z$, then we can get $x \approx^k z$. For example, again in Figure 1, nodes *11,13* are *1-bisimilar*, while *11,13* are not *2-bisimilar* and nodes *12,13* are not *1-bisimilar*.

4 XML Path Indexes

In this section, we start our survey of XML path indexes for answering XML queries. We mainly discuss the approaches described in [**?**, **?**, **?**, **?**, **?**]. With these structures, a query can efficiently locate the results without traversing the data graph or the whole indexed message.

4.1 DataGuide

As the nascence in this area, DataGuide [?, **?**] is a concise, accurate, and convenient summary of semistructured databases. DataGuide could be dynamically generated without reference to DTD (for *Document Type Definition*) or XML Schema [**?**] of XML documents. Nevertheless, DataGuide requires a powerset construction over the underlying database, which in the case of large cyclic graphs can be of exponential cost.

4.1.1 DataGuide

The DataGuide [?] in its simplest form is a label path index, i.e. it supports path matching, but not keyword matching. Its main data structure is a tree (or graph when indexing graph documents) consisting of all label paths which occur in the indexed document collection. An important characteristic of the DataGuide, is that each label path from the document collection appears exactly once in the index tree. This resembles closely a schema tree. The only difference is that DataGuide is not required to be tree, it is an OEM object. Instead of matching a given query path in the document tree, the index tree is used which is usually much smaller than the former and therefore resides in main memory. Next we give the formal definition of DataGuide proposed in [?].

DEFINITION 10. *(DataGuide:)* A DataGuide for an OEM source object s is an OEM object d such that every label path of s has exactly one data path instance in d, and every label path of d is a label path of s. ■

Figure 2 shows a DataGuide for the source XML database shown in Figure 1. With DataGuide, we could quickly check whether a given label path of length n exists in the original database by scanning at most n objects in the DataGuide. For example, we need only examine the outgoing edges of objects 0, 1 and 3, in Figure 2 to verify that the path */site/people/person* exists in the database. An extent in a DataGuide denotes all objects reachable by a given label path. For instance, the extent of node 7 in Figure 2 is {&7, &8} in Figure 1 reachable by label path */site/people/person*.

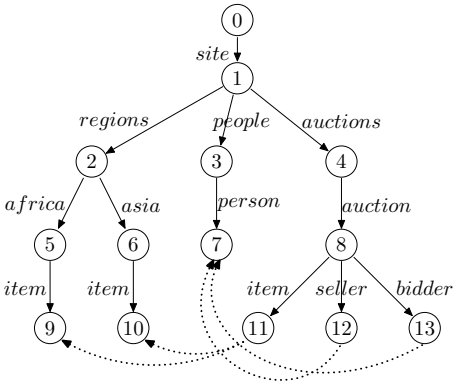


Figure 2. DataGuide

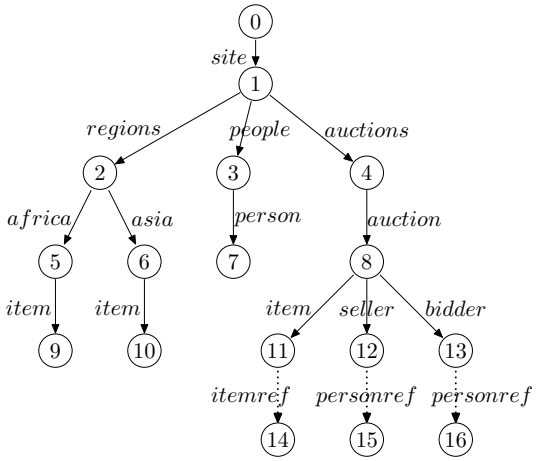


Figure 3. Strong DataGuide

4.1.2 Strong DataGuide

Strong DataGuide is a class of DataGuide which is more restrictively. In Strong DataGuide, each label paths that share the same extent in the DataGuide is exactly the set of label paths that share the same target set in the source. Formally, it is defined as:

DEFINITION 11. *(Strong DataGuide:)* Consider OEM objects s and d, where d is a DataGuide for a source s. Given a label path l of s, let $T_S(l)$ be the extent of l in s, and let $T_d(l)$ be the extent of l in d. Let $L_s(l) = \{m | T_s(m) = T_s(l)\}$. That is, $L_s(l)$ is the set of all label paths in s that share the same extent as l. Similarity, let $L_d(l) = \{m | T_d(m) = T_d(l)\}$. That is, $L_d(l)$ is the set of all label paths in d that share the same extent as l. If, for all label paths l of s, $L_s(l) = L_d(l)$, then d is a *strong* DataGuide for s. ■

For example, Figure 2 is not a strong DataGuide for Figure 1. Set the label path $l = site.people.person$. The source extent $T_s(l)$ is {7, 8}. In the source, $L_s(l)$ is {*site.people.person*}. In the DataGuide, however, $L_d(l)$ is {*site.people.person, site.auctions.auction.seller.personref, site.auctions.auction.bidder.personref*}. Since $L_s(l) \neq L_d(l)$, the DataGuide is not strong. We give an example of strong DataGuide in Figure 3.

DataGuide requires a powerset construction over the underlying database, which in the worst case can be of exponential cost. Evaluating query paths with DataGuide is rather easy, just start from the root, navigate through the edge until getting the extent needed.

4.2 T-Index

T-index, or *template index*, is proposed to answer query for specified path template. T-index allows us to trade space for generality. DataGuide requires a powerset construction over the underlying database, which in the worst case can be of exponential cost. While by contrast, T-index rely on the computation of bisimulation relation. And the size of T-index associated to a single regular expression is at most linear in that of the database.

The XML database in Figure 1 is a little complex to show the T-index. We use, therefore, a simple case to show the structure of T-index. Figure 4(a) is a simple XML data graph as shown in [?].

4.2.1 1-index

1-index is the most simple template index for efficiently answering $q \in inst(\boxed{P}\ x)$. It is based on the equivalence class of $\approx$. A equivalence class of node u is represented as $[u]$. Now we give the definition of 1-index.

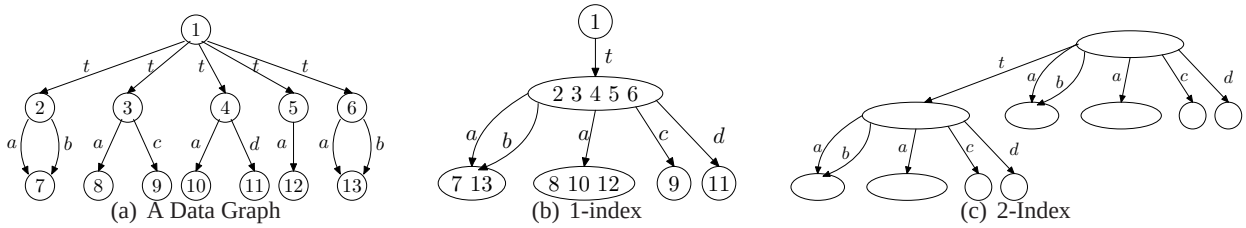
DEFINITION 12. *(1-index:)* Given a data graph G_x and a refinement $\approx$, the 1-index $I(G_x)$ is a rooted, labeled graph defined as follows. Its nodes are equivalence classes $[v]$ of $\approx$; for each edge $v \xrightarrow{a} v'$ in G_x there exists an edge $[v] \xrightarrow{a} [v']$ in $I(G_x)$; the roots are $[r]$ for each root r in G_x. When G_x is clear from the context we omit it and simply write I. ■

Figure 4(b) is an example of 1-index of Figure 4(a). Evaluating query paths on 1-index is easy. First get the set of nodes in $I(G_x)$ that satisfy the query path, represented as $\{s_1, s_2, \ldots, s_k\}$. The answer of the query on G_x is $extent(s_1) \cup extent(s_2) \cup \ldots \cup extent(s_k)$. We omit the proof of this algorithm, interesting reader could reference to [?].

4.2.2 2-index

2-Index is proposed to answer the template path $*x_1\boxed{P}\ x_2$. Here we are only interested in pairs of nodes (x_1, x_2), so we first define $L_{(v,u)}$ as the path in G_x existing a path from v to u. We also use $[(v, u)]$ to denote the equivalence class (v, u) using the refinement $\approx$. Now we give the definition of 2-Index.

DEFINITION 13. *(2-Index (I^2):)* Its nodes are equivalence classes, $[(v, u)]$, of $\approx$; the roots are all the equivalence classes of the form $[(x, x)]$; finally, for each edge $u \xrightarrow{a} u'$ and each node v in G_x there is an edge $[(v, u)] \xrightarrow{a} [(v, u')]$.We store I^2 as two logical components: the graph itself and the extent of s as $[(v, u)]$, for each node s representing the equivalence class $[(v, u)]$. ■



(a) A Data Graph (b) 1-index (c) 2-Index

Figure 4. (a) A data graph, (b) its 1-index, (c) its 2-Index

Figure 4(c) is an example of 1-index of Figure 4(a). Query evaluation with 2-Index proceeds similarly to that of 1-index, with just a small modification: to compute $*x\boxed{p}\ y$, we compute the query path $\boxed{P}$ y on I^2 and take

the union of the extents. Note that this saves the $*$ search, while we may need to start at several root in $\hat{P}$ even if there is only a few.

4.2.3 T-index

Now we discuss the general T-index. The idea of T-index is that rather than indexing all the paths, it is more convenient to index those having high percentage accessed queries. Another observation is the case that most of the information in the database has a fixed, pre-defined structure and only certain components are irregular.

We fix the path template $t = T_1 x_1 T_2 x_2 \dots T_n x_n$, where each of the $T_i's$ is either a path expression or a place holder $\boxed{P}$ or $\boxed{F}$. T-index is built to answer the queries $q \in inst(t)$. Note that T-index both generalizes and specializes 1-index and 2-index in certain way and 1-index and 2-index are particular cases of T-index. We first generalize the $\approx$ to the language-equivalence relation, $(v_1, \dots, v_i) \approx (u_1, \dots, u_i)$, iff $T_{v_1,\dots,v_i}(DB) = T_{u_1,\dots,u_i}(DB)$, and denote the equivalence class of $(v_1, \dots, v_n)$ as $[(v_1, \dots, v_n)]$. Now we give the definition of T-index,

DEFINITION 14. *(T-index:)* Given $\approx$, the T-index $I^t(DB)$ for t is the following rooted, labeled graph:
$Nodes$:- The nodes include all the equivalence classes(w.r.t $\approx$)$[(v_1, \dots, v_n)]$, $i = 1, n$. Also, for each such class we introduce an additional new node which we denote $[(v_1, \dots, v_n)]^\$$
$Edges$-For each i-tuple there is an edge labeled \$ from $[(v_1, \dots, v_n)]^\$$ to $[(v_1, \dots, v_n)]$, $1 \leq i < n$. Additionally, each T_i in the template t introduces some edges, depending on its structure.
Root Nodes- The roots are all the nodes $[(v)]$ where v is a root of DB.
terminal nodes- there are all nodes of the form $[(v_1, \dots, v_n)]^\$$.
Finally we remove all nodes not reachable from a root or not having an outgoing path to a terminal node, and associate with each terminal node $[(v_1, \dots, v_n)]^\$$ the extent containing all tuples in $[(v_1, \dots, v_n)]$. ■

The concrete cas of T-index depends on the path templates adopted so we omit the example. However, we could deduce the structure of T-index from the structures of 1-index and 2-index shown.

4.3 Index Fabric

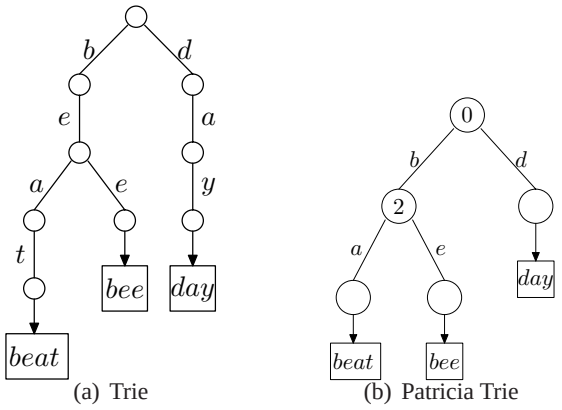
Index Fabric [?] is another approach of path indexes. Based on Patricia trie [**?**,**?**,**?**] to index strings, Index Fabric is optimized to provide the efficiency and flexibility for both general queries and specific access paths.

Before discussing Index Fabric, we first introduce the idea of Patricia tree in brief. Figure 5(a) is an example trie indexing strings *beat*, *bee* and *day*; Figure 5(b) is corresponding Patricia trie by compressing the trie structure. The nodes are labeled with their depth: the character position in the key represented by the node.

The first question is how to fill the gap between string and XML path expression. The solution is to use *designator dictionary* to assign each unique element tagname a character called *designator*. The efficiency of *designator dictionary* profits from the case that the number of element tagname in an XML document is always not large. In fact, Patricia tries are still unbalanced, main memory structures that are rarely used for disk-based data. Inspired by B-trees, Index Fabric uses a layered method based on Patricia trie, which is balanced and optimized for disk-based access.

In constructing the Index Fabric, the basic Patricia trie is divided into block-sized subtries, and these blocks are indexed by a second trie, stored in its own block. We generate a new horizontal layer to the higher level trie. If the new horizontal layer is still too large to fit in a single disk block, we recursively generate a new layer to index it. An Index Fabric example of Figure 1 is shown in Figure 7. Table 6 gives the designators for the data graph in Figure 1. For convenience, we consider the XML data graph as a data tree structure here and omit all the *IDREF* labels in Figure 1.

The trie in layer 1 is an index over the common prefixes of the blocks in layer 0, where a common prefix is the prefix represented by the root node of the subtrie within a block. In Figure 7, the common prefix for each block is



(a) Trie (b) Patricia Trie

Figure 5. (a) Trie, (b) Patricia Trie

Tagname	Designator
$site$	α
$regions$	θ
$africa$	β
$asia$	γ
$item$	δ
$people$	η
$person$	π
$auctions$	υ
$auction$	χ
$seller$	μ
$bidder$	ψ

Figure 6. Designator Dictionary

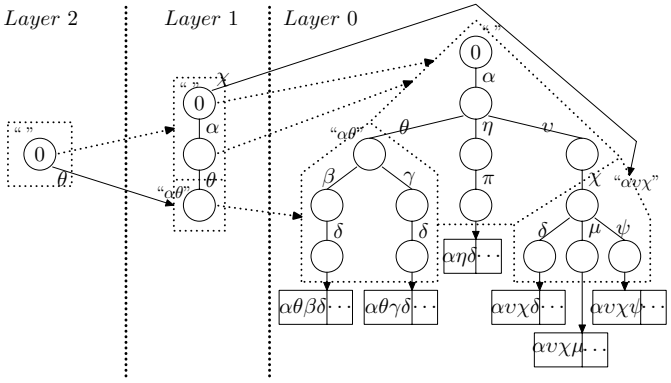


Figure 7. Index Fabric

shown in "quotes". Similarly, layer 2 indexes the common prefixes of layer 1. The index can have as many layers as necessary; the leftmost layer always contains one block. In Figure 7, the labeled link edge likes normal edges in a trie, except that it connects nodes between different horizontal levels. The unlabeled link edge connects a node in one layer to a block with a node representing the same prefix in the next layer. Figure 7 just show the paths called *raw paths* for XML database. In fact, we could encode any path and insert it into the Index Fabric, this kind of path is called *refined path*.

The search process begins in the root node of the block in the leftmost horizontal layer, examines one block per layer, and always examines the same number of layers. If the blocks correspond to disk blocks, this means that the search could require one I/O per layer, unless the needed blocks in the cache. The key benefit of using Patricia structure is that keys are compactly stored, and many keys can be indexed per block. In practice, an index storing a large number of keys (e.g. a billion) requires only three layers: layer 0 must be stored in disk but layers 1 and 2 can reside in main memory. So key lookups require at most one index I/O (in addition to data I/Os).

4.4 A(k)-Indexes, D(k)-Indexes and M(k)-Indexes

Indexes such as DataGuide and T-index are designed to answer all path queries accurately, which need encode long, seldom-queried paths, leading to increased size and complexity with little added value. The key observation is that not all structure is interesting. In particular, long and complex paths tend to contribute disproportionately to the complexity of an accurate structural summary. Next we will introduce a series of indexes built on the idea

of k-bisimilarity.

4.4.1 A(k)-indexes

Based on k-bisimilarity, A(k)-indexes [**?**] is a family of efficient, flexible data structures capable of serving as structural summaries of graph-structured data. The key observation behind A(k)-index is that not all structure is interesting in particular long and complex paths. For paths no longer than k, A(k)-index could get exact answers. While for paths longer than k, the A(k)-index becomes approximate, thus could utilize the similarity of short paths to reduce the size of the structure.

A(k)-indexes is based on the idea of k-bisimilarity, in which nodes are grouped based on local structure, i.e., the incoming paths of length up to k. A sequence of A(k)-indexes examples of Figure 1 are shown in Figure 8. For convenience, we omit all the *IDREF* labels again.

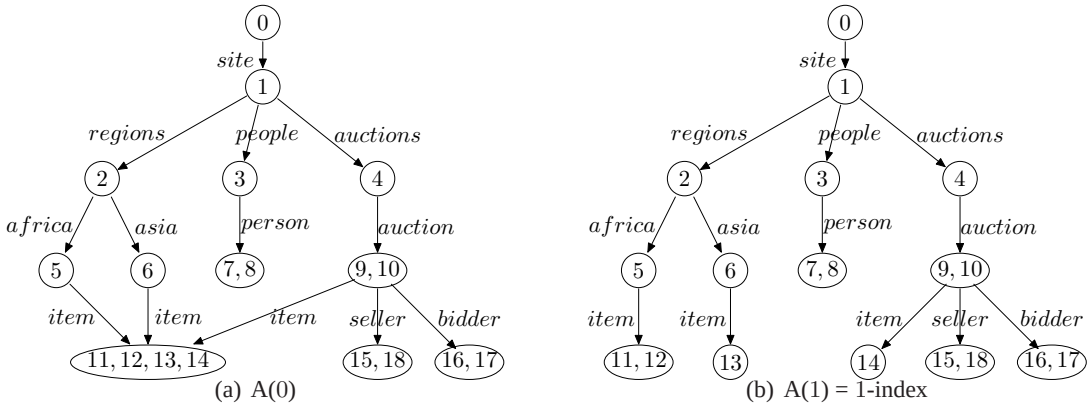


Figure 8. A(k)-indexes

A(0)-index and A(1) are based respectively on 0-bisimilarity and 1-bisimilarity. In this simple case, the 1-bisimulation is actually the maximal bisimulation-based 1-index. The algorithm to construct the A(k)-index is a variant of he standard algorithm [**?**] for computing the bisimulation.

A(k) could exactly answer path queries no longer than k. For example, Figure 8(a) could give the exact answers of queries *//item* and *//seller*. While Figure 8(b) could give exact answers of queries like *//auction/item* or *//people/person*. The evaluation of an approximate path expression R on A(k)-index is to use the (forward or backward) automaton evaluation strategy, adding the nodes in *ext[B]* rather than B itself to the result set when B is accepted by the automaton. Since A(k)-indexes is safe, which means that the A(k)-index result set for R is always a superset of the target set. We need validate the nodes of M which have possible false positives against the original data graph. The validation is handled by a reverse execution of the automaton on the data graph starting with each node in M. Moreover, some optimization techniques to avoid needless validation are proposed [?] to improve the efficiency.

4.4.2 D(k)-indexes

D(k)-index is also based on local similarity and built on previous work, 1-index and A(k)-index. 1-index and A(k)-index cannot adjust their index graphs according to the different structure complexity of the equivalence classes required by the query load, because they assume the uniformity of query patterns. In contrast, D(k)-index is an adaptive structural summary for graph-structured data by assigning different bisimilarity requirements to different types of data nodes according to the query load, which can dynamically adjust its structured optimally to achieve reduced index size and improved performance.

Before defining the D(k)-index, we first give the symbol k_A. Given an index node A, k_A is the local similarity required by queries. The value of k_A can be obtained by mining the current query load and it should guarantee that the majority of queries accessing A are less than or equal to k_A in length. So we could directly perform most queries on the index graph without the validation process, with which to eliminate the potential workload to improve the system performance. Now we give the definition of D(k)-index,

DEFINITION 15. *(D(k)-index:)* The D(k)-index is the index graph based on local bisimilarity that satisfies the condition that for any two nodes n_i and n_j, $k(n_i) \geq k(n_j) - 1$ if there is an edge from n_i to n_j, in which $k(n_i)$ and $k(n_j)$ are n_i and n_j's local similarities, respectively. ■

According to the definition, 1-index and A(k)-index are both special cases of D(k)-index. In D(k)-index, the local similarity of the parent plus 1 cannot be less than the local similarity of its child. The simplest case of D(k)-index is constructed by label split with the local similarity of each index node equal to 0.

We could see Figure 8(a) as a D(k)-index with all nodes having similarity 0. Also we could see Figure 8(a) as a D(k)-index with all nodes having similarity 1, or only nodes $item$ having similarity 1 while others having similarity 0. The search strategy of D(k)-index is similar to that of A(k)-index but efficient in that it could avoid potential validation operation which profits from its construction and mining query loads.

4.4.3 M(k)-index

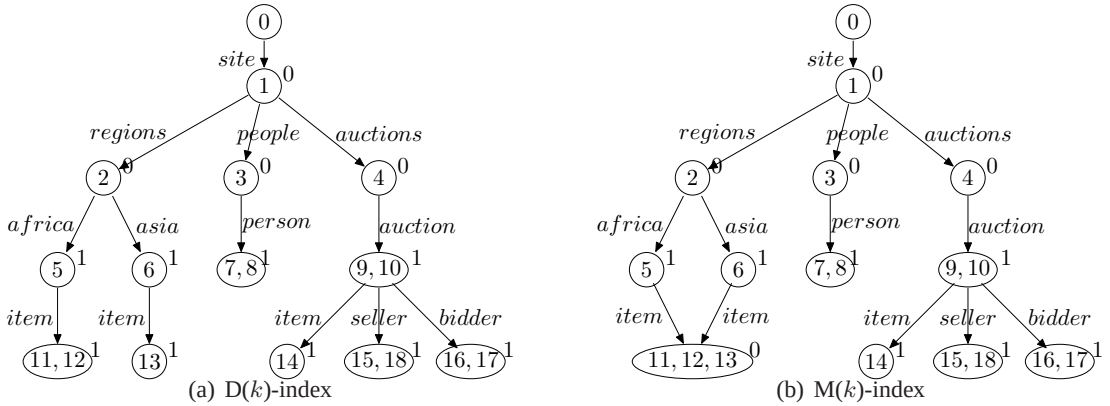
M(k)-index is based on D(k)-index. The general approach of the D(k)-index is flexible and powerful, nevertheless, the design still has several limitations to be overcome, mainly comes from the over-refinement problems. For instance, the over-refinement of irrelevant index nodes, over-refinement for irrelevant data nodes, over-refinement due to overqualified parents and single resolution per node.

To overcome the first two of the above limitations, they proposed M(k)-index (for "mixed-k"). Like A(k)-index and D(k)-index, the M(k)-index also uses the k-bisimilarity equivalence, the difference is that M(k)-index allows different k values for different index nodes having the same tagname. Compared with D(k)-index, the M(k)-index is not over-refined for irrelevant index or data nodes. Therefore, M(k)-index has a smaller size without sacrificing support for any frequently used path expressions. To overcome the last two limitations, M(k)-index could be further refined to get M*(k)-index, which consists of a collection of M(k)-indexes whose nodes are organized in a partition hierarchy, allowing successively coarser partitioning information to co-exist with the finest partitioning information required. In conception, the M*(k)-index consists of a sequence of component indexes I_0, I_1, ..., I_k with different resolutions. Each component can be regarded as an M(k)-index that supports the frequently used path expressions as much as possible, abiding by the restriction that the maximum local similarity in component I_i is i. Figure 9 is the comparison of D(k)-index and M(k)-index. The number on the upper-right of each node is the value k of bisimularity. As shown in Figure 9(a), all index nodes of *item* are 1-bisimulation, while in Figure 9(b), the M(k)-index case, some of the index nodes of *item* are 0-bisimulation while some are 1-bisimulation.

To answer a simple path expression l using the M(k)-index, we first find the target set of l in the index graph, i.e., the set of index nodes with l as an incoming path. Then for each index node v in the target set, if $v.k \geq length(l)$, we return all of *v.extent* as the result set. Otherwise, we must validate the nodes in *v.extent* and only return to the user those that really do have l as an incoming path in the data graph.

5 Comparison of the Approaches

Before comparing these approaches based on storage space, construction cost and update cost, we first show the data representation of the index structures. Without considering the practical implementation, all the index structures in this paper could be viewed as two parts: the structural summary of the database, and an extent of



(a) D(k)-index (b) M(k)-index

Figure 9. Comparison of D(k)- and M(k-index)

each node for simple paths to accessed data in the database. The extent could be viewed as the secondary index of the database. Among the approaches, Index Fabric is a little different because it aims at indexing the compressed label while not the original label name. In its simplest form Index Fabric is a Patricia Trie indexing combined label and character strings. As such it is suitable for tree databases and possible directed labeled graphs in general, but not for cyclic graphs. As both structure and content are kept in the same data structure, there is no need for a secondary content index. The layered Index Fabric is a hierarchy of Index Fabrics where each pair located at adjacent layers is interconnected by two different kinds of link: labeled far links, and unlabeled direct links. On each layer, the Index Fabric is divided into blocks which may correspond to disk blocks when the index is stored on a secondary device. Both the simple and the layered Index Fabric assume a single database graph rather than separate document graphs.

5.1 Construction

[**?**] proved that creating a DataGuide is equivalent to conversion of a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA), a well-studied problem in [**?**]. When the source data is a tree, the conversion is in linear time. In the worst case, nevertheless, conversion of a graph structured database may require exponential time in the number of objects n and edges m in the source.

The construction of basic Index Fabric is easier than DataGuide because the basic Index Fabric is a tree, which only takes linear time $O(m)$, where m is the number of edges in the source. There is no accurate construction cost for Index Fabric with many refined paths, which depend on the practical case. Furthermore, the higher horizontal layers are easy to construct based on lower layers.

The construction cost of 1-index is also in linear time when the source data is a tree. However, in the worst case, the 1-index graph can never be larger than the data graph. 1-index can be constructed in $O(m \lg n)$ time using Paige and Tarjan's algorithm [?], where n is the number of nodes and m is the number of edges in the data graph. But for 2-index, the construction cost is quadratic time in the worst case. For T-index, it depends on the actual path template t selected.

The algorithm to compute the A(k)-index is a variant of the standard algorithm [?] for computing the bisimulation. The A(k)-index can be computed in $O(km)$ time, where m is the number of edges in the data graph G_x and k is the local similarity required. The A(k)-index is constructed once the k-bisimulation partition is computed.

The construction of D(k)-index begins with the simplest index graph, the A(0)-index, in which each node represents a tagname and label represents the possible relation among different nodes. [?] presented a broadcast algorithm to compute the actual local similarities of labels in the D(k)-index. The broadcast algorithm takes $O(l)$ time to construct the D(k)-index from A(0)-index, in which l is the number of edges in the A(0)-index. While the

conversion from source data to A(0)-index need linear time $O(m)$ where m is number of edges in the source data. As a whole, the D(k)-index construct algorithm takes $O(l + m)$ time.

The construction of M(k)-index is based on A(0)-index and recursively refine the index using the frequently used path expressions from queries. The practical construction cost depends on the concrete frequently used path expressions used so it is uncertain.

5.2 Storage Space

For DataGuide, the space requirement is linear of tree structure source data $O(m)$, where m is the number of edges in the source data graph; for graph structure, the worst case is exponential in the size of the database. The DataGuides size depends dramatically on the structure of the documents. If they contain only few distinct paths and no cycles, the index structure may be held in main memory. On the other hand, deeply nested documents with long-distance backlinks deteriorate both storage overhead and indexing time.

By virtue of Patricia Trie compression, the IndexFabrics storage requirements remain modest even for large databases, so that many refined paths can be indexed additionally at almost no expense. Indeed inserting long paths into the index does not immediately affect the index size and thus its performance. Compression also keeps the hierarchy shallow, removing chains of nodes with a single child each. However, in real-world databases deeply nested trees with only one node per level are rare, so path compression is unlikely to contribute much to an overall reduction of the index size. Thus the space requirement of Index Fabric is proportional only to the number of nodes n which is $O(pn)$, here p is a parameter related to the structure of source graph data. Furthermore, the sophisticated paging strategy of the Index Fabric (which might also be applicable to other navigational indices) is a considerable advantage when indexing huge document collections.

Unlike strong DataGuide, in which the extents may overlap. The storage for 1-index can, therefore, be smaller than for strong DataGuide. The storage of 1-index consists of the graph I and the sum of all extents. Compared to the DataGuide which needs exponential size in the worst case, 1-index only need linear size $O(m)$ of database, where m is the number of edges in the source data graph. And for 2-index, the two parts: graph and extent, both are now (at worst) quadratic $O(m^2)$ in the size of DB. T-index is a general idea so its size depends on the practical case of path template t adopted. Experiment with the data from the Open Directory Project (ODP) [**?**] shows that, the size of 1-index turns out to be about 45% of the data graph size.

The storage space of A(k)-index is $O(m)$, where m is the number of edges in the G_x. As the case the 1-index, the size of A(k)-index has an upper bound dependending only on the number of distinct tags and the length of the longest simple path (i.e., a path without cycles) in the data, independent of the size of the data. One worst case is when each data node has a unique tag, which means that there is absolutely no structure in the data. And in this kind of worst case, the size of A(k)-index equals to the size of data. Experiment with a subset of the Internet Movie Database [**?**] shows that, the A(3)-index is much smaller than the 1-index with 62% fewer nodes.

The other two approaches, D(k)-index and M(k)-index are all based on bisimilarity and in the same complexity A(k)-index, the storage space of D(k)-index and M(k)-index are also $O(m)$. Now we consider the M*(k)-index. Because the M*(k)-index consists of a sequence of M(k)-indexes, it may appear that the M*(k)-index needs more space than the M(k)-index in order to maintain partitioning information at multiple resolutions. However, the implementation of M*(k)-index can be made much more space-efficient than its logical representation by exploiting the fact that many index nodes and edges remain unchanged from one component index to the next. Moreover, their experiments on XMark [?] and NASA [**?**] datasets reveal that the M*(k)-index is actually smaller than the M(k)-index in most cases because the M*(k)-index completely avoids over-refinement due to overqualified parents.

5.3 Update

The update algorithm given in [?] is very similar to the one used for building DataGuides. It allows to insert entire subgraphs into the database, which is then updated recursively. This is valuable e.g. when a new document is added to the collection. Some persistent data structures initialized during index creation guarantee that only those parts affected by the modification are re-examined. Updates of the DataGuide, although incremental, may be costly because individual document nodes can be referenced at multiple points in the index structure. Hence more than one path may need to be examined at the risk of expensive I/O accesses to secondary storage devices. Note that since content is not represented in the index, modifying atomic values in the database does not result in an update of the DataGuide (but maybe of the content index).

Since the hierarchal structure of Index Fabric is balanced, in the worst case an insertion may cause all blocks in the horizontal path leading to the insertion point to be split. Analogously, removing paths from the index can result in block merging, which is not discussed in the paper. In any case, the Index Fabric can be adapted to database changes incrementally as long as no additional constraints are introduced by node identification schemes.

In fact, in the series of T-indexes, the 1-index is the most universal and concise compared with 2-index and T-index, therefore, we mainly discuss the update for 1-index here. The update algorithms of 2-index and T-index are all based on that of 1-index. Because 1-index is based on graph bisimulation, the update algorithm of DataGuide can not be generalized to apply in this context. Update algorithms for 1-index are presented in [**?**]. The algorithm in [?] could find the new 1-index without having to recompute it from scratch when adding a subgraph, also it support incremental update of the data.

To overcome the big size of 1-index for some data sets, A(k)-index was introduced. The update algorithm is a variant of 1-index update algorithm, supporting the subgraph adding and incremental update. When a new edge is added to the A(k)-index, it creates a new index node and recursively checks if the new node's child satisfies k local similarity. Stops if yes and partitions the extent of the target node such that the data nodes in the extents are actually k-bisimilar. It has been shown that a small change in a graph can trigger large changes in the 1-index and A(k)-index. In the worst case, the edge addition needs to touch $O(n + m)$ nodes and edges in the data graph.

As to the update algorithms of 1-index and A(k)-index, the update algorithm for D(k)-index [?] also supports two kinds of updates: the addition of a subgraph and the addition of a new edge. In contrast to A(k)-index, the D(k)-index update algorithm for edge addition is more efficient in that it simply lowers the local similarities of the affected index nodes. There is no update algorithm proposed for M(k)-index but we could treat it as an invariant of D(k)-index.

Table **??** summarizes the path index approaches discussed in this paper. Generally, we seek upper bounds on the complexity, because everybody likes a guarantee. Here, n is the number of nodes, m is the number of edges in the source data graph, l is the number of edges in the A(0)-index, k is the local similarity required and p is the number of regular path expression $\boxed{P}$ in template path t, p is an uncertain parameter related to the structure of source graph data G_x. We don't show the construction cost of M(k)-index in the table. The reason is that it is uncertain, depending on the frequently used path expressions used for refinement. All approaches summarized support incremental update *incr.*, so we omit it in the table. Whereas, when some approaches adopt some numbering scheme, such as region code [**?**], adopted by B^+-tree [**?**] and XR-tree [**?**] for structural join [?]. The reason is that region encode could not well serve the update, which maybe renumber all nodes when inserting new nodes to guarantee the correct node order. They could not, hereby, support incremental update.

5.4 How to select ?

In this section we analyze which approach to select under various applications. Notwithstanding that in common sense, the later approaches seem to be natural choices because they are based on and evolve the formers. Note that, however, the design of later approaches are based on assumptions, which are rational under some cases, but

Index Name	*Data Structure*	*Construction*	*Storage*
DataGuide	graph	$exp(m+n)$	$exp(m)$
1-index	graph	$O(m \lg n)$	$O(m)$
2-index	graph	$O(m^2 \lg^2 n)$	$O(m^2)$
T-index	graph	$O(m^p \lg^p n)$	$O(m^p)$
Index Fabric	tree	$O(m)$	$O(pn)$
A(k)-index	graph	$O(km)$	$O(m)$
D(k)-index	graph	$O(l+m)$	$O(m)$
M(k)-index	graph		$O(m)$

Table 2. Comparison of Index Approaches

not absolute.

First we classify the regular path expression. The path expression could be simply classified as single path expression which contains only one path and branch path expression which contains twigs. The single path expression could be even classified as absolute path expression and relative path expression. Absolute path expression only includes the parent-child relation "/", while relative path expression include both parent-child relation "/" and ancestor-descendant relation "//". Hence the absolute single path expression is the simplest, and branch path expression is the hardest to dispose. Most of the approaches in this paper are the solutions for simple path expression. There are, but no very efficient, solutions for branch path expression now. The reason is that the combination of branch path for a graph itself is a NP problem, index all these cases is even harder. In general, the query for branch path expression could be resolved by splitting it into single path expressions for searching and then join all the intermediate results. Another strategy is to traverse the subtree in index structure to perform the "//" operation.

The DataGuide and 1-index are the approaches for general cases. Without any statistics and the length and type of queries is unpredictable, they can be used. Navigation of absolute path expression is directly over the indexes. For relative path expression, one common trick is query rewriting technique which converts relative path expression to absolute path expressions utilizing the DTD or schema information. 2-index and T-index, in fact, are complex and seldom used. Index Fabric is a compact structure and well stored for searching. It could support all the queries even branch path expressions as refined paths. One basic assumption of Index Fabric is that there is a huge or infinite size designator dictionary for characters. For example, if the designator dictionary only includes character for 'a' to 'z', when the $27th$ tagname is to be designated, you cannot encode it as 'ab' or similar, which could make mistake in searching. Anyway, the size of designator dictionary is finite, so if there are not much tagnames in a graph and not much refined path, Index Fabric is a good choice. The A(k)-index series, including D(k)-index and M(K)-index, based on the assumption that the length of most path expressions is short. So the number k could be a small integer as 3 or 4 to efficiently support most queries with less storage overhead. In some applications, if most queries are all not long, A(k)-index is a option. When there are many long single path expressions, in another way, the query results of A(k)-index are not safe and need to be refined with source data which is obviously time consuming. And in this case, A(k)-index series should not be used. Furthermore, if we have already many statistics for mining frequently used queries, D(k)-index and M(K)-index could be selected to accelerate the search.

6 Related Works

Much research has been done on XML path indexes in recent years. Forward and Backward-Index (F&B-Index) [**?**], defined directly from ideas proposed in [**?**], can be viewed as a covering index for branching path expression queries. The full F&B-Index (covers all path expression) is too large so they optimize it based on specially chosen subsets of path expressions. APEX [**?**], an adaptive path index for XML data which consists

of two structures: a graph structure and a hash tree. In contrast to previous methods which utilize frequently used paths to improve query performance, APEX adopts data mining algorithm to summarize paths that appear frequently in query workload. Thereby, APEX is efficient in processing partial matching queries, and also, low update cost by incremental update. Another novel design is ViST [**?**], a dynamic index method for searching XML documents. By representing both XML documents and XML queries in structure-encoded sequences, it shows that querying XML data is equivalent to finding subsequence matches, and uses tree structure as the basic unit of query to avoid expensive join operations. ViST could also support dynamic update operation. PRIX is introduced in [**?**] for twig pattern queries, which indexes and queries XML using Prüfer [**?**] sequences. PRIX first transforms each XML document into a sequence of labels by Prüfer's method that constructs a one-to-one correspondence between trees and sequences. In query processing, the twig pattern is also transformed into its Prüfer sequence and perform query operation by efficient subsequence matching of Prüfer sequence. Furthermore, there is still a lot of work in this area, interesting reader could reference to [**?**, **?**, **?**, **?**, **?**].

7 Conclusion

As shown in this survey, answering XML queries using path indexes raise many challenges. Many issues are proposed in the literature, we introduce some important approaches following the idea of DataGuide or bisimulation from year 1997 to now. All these issues and plenty of experiments have shown that path indexes could greatly increase the performance of XML queries and consequently, decrease the system response time.

Although many good index structures have been proposed, there are still many issues remain open in this realm. The current approaches perform well in both long and short path query, however, most approaches are still not optimized for disk-based structure as B^{+}-$tree$. Another direction is the path index for branch query. although the F&B-Index [?] can be viewed as a covering index for branching path expression queries. The full F&B-Index (covers all path expression) is too large to use. And they optimize it for chose subsets of path expressions. While this issue deserves further research. Another direction is query including many "//" operation. For instance, "//a//b//c", which is still expensive to perform with current approaches.

In conclusion, there are many issues to be resolved and many issues to be appeared along with the development of both XML and related applications. The research for answering XML queries using path indexes will continue to be a hot topic and become more and more valuable in many applications.

Acknowledgement

This work is partially supported by the CUHK strategic grant (numbered 4410001).